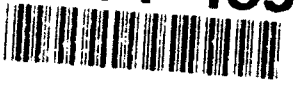


Fault-Tolerant Key Distribution*
(Preliminary Version)

Michael Reiter
Kenneth Birman
Robbert van Renesse

TR 93-1325
January 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Office of Naval Research (ONR) under grant number N00014-92-J-1866, and by grants from GTE, IBM and Siemens, Inc. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views or decisions of the ONR.

Fault-Tolerant Key Distribution*
(Preliminary Version)

Michael Reiter
reiter@cs.cornell.edu

Kenneth Birman
ken@cs.cornell.edu

Robbert van Renesse
rvr@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, New York 14853

January 4, 1993

Abstract

Many *authentication* or *key distribution* protocols have been proposed to distribute cryptographic keys for secure communication in open networks. These protocols often employ trusted authentication and time services whose corruption or failure could result in security breaches or prevent correct principals from establishing secure communication. In this paper, we describe the design and implementation of authentication and time services that securely and fault-tolerantly support key distribution. By using replication only when necessary, and introducing novel replication techniques when it was necessary, we have constructed these services to be easily defensible against malicious attack. Moreover, the transient unavailability of even a substantial number of servers does not hinder key distribution between correct principals or expose protocols to intruder attacks. We also describe how these services function as the foundation for a more comprehensive security architecture that we have implemented for fault-tolerant systems.

1 Introduction

In open networks, an intruder can attempt to initiate spurious communication in two ways [VK83]: it can try to initiate communication under a false identity, or it can replay a recording of a previous initiation sequence. Many *authentication* or *key distribution* protocols have been proposed to protect against these attacks (e.g., [NS78, DS81, OR87, SNS88]). These protocols allow principals (e.g., computers, users) initiating communication to verify each others' identities and the timeliness of the interaction. Most also arrange for the involved principals to share a secret cryptographic key by which subsequent communication can be protected, or to possess each others' public keys, by which communication can be protected or a shared key can be negotiated.

*This work was supported by the Office of Naval Research (ONR) under grant number N00014-92-J-1866, and by grants from GTE, IBM, and Siemens, Inc. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views or decisions of the ONR.

Authentication protocols typically employ a trusted service, commonly called an *authentication service* [NS78], to counter the first type of attack. In shared key protocols, the authentication service normally shares a key with each principal and uses these keys to distribute other shared keys by which principals communicate. In public key protocols, the authentication service usually has a well-known public key and uses the corresponding private key to certify the public keys of principals. Public key authentication services are also called *certification authorities* [BAN89].

A predominant technique to detect replay attacks in authentication protocols is to incorporate into each protocol message the time at which the message was generated; the message is then valid for a certain *lifetime*, beyond which it is considered a replay if received [DS81]. Timestamp-based replay detection has been used in several systems (e.g., [SNS88, TA91]) and is often preferable to challenge-response techniques [NS78], because it results in fewer protocol messages and less protocol state. However, this approach requires that all participants maintain securely synchronized clocks. In practice, clock synchronization is usually achieved via a *time service*, as in [GZ84, Mil89].

The dependence of authentication protocols on authentication and time services raises troubling security and availability issues. First, the assurances provided by authentication protocols directly rely on the security of these services, and thus these services must be protected. Second, the unavailability of these services may prevent correct principals from establishing secure communication, or even open security "holes" that could be exploited by an intruder. For instance, the unavailability of a time service could result in clocks drifting far apart, thereby exposing principals to replay attacks.

To increase the likelihood of these services being available, they could be replicated. However, in many environments this is dangerous, because replicating data or services makes them inherently harder to protect [HT88, LABW91, Gon93]. That is, there is an apparent conflict between security and availability when implementing core security services, as it is generally more difficult, or at least requires more resources, to protect a replicated service than it is to protect a nonreplicated one.

We have developed techniques to reconcile this conflict. Specifically, we have constructed authentication and time services, and ways to use them, that securely and fault-tolerantly support key distribution. By using replication only when necessary, and introducing novel replication techniques when it was necessary, we have constructed these services to be easily defensible against attack. And, the transient unavailability of even a substantial number of servers does not hinder key distribution between principals or expose protocols to intruder attacks. Client interactions with the services are simple and efficient, and the services can be used with many different authentication protocols.

These services form the core of a more comprehensive security architecture that we have implemented for fault-tolerant distributed systems. The architecture provides tools for building fault-tolerant applications that remain correct despite malicious site corruptions and network attacks. Although not tied to this system, our services are well-suited to support this architecture due to their fault-tolerance features. After describing our services, we will discuss how they are used in this larger security architecture.

2 The time service

The security risks of clock synchronization failures in authentication protocols are well-known [DS81, Gon92], and the need for a *secure* time service has been recognized in several systems (e.g., [Mil89, BM90]). However, the case for a *highly available* time service is not as clear. It is true that an extended period of unavailability might cause principals to have increasingly disparate views of real time. But, this in itself need not result in security weaknesses or inhibit communication too quickly. In evidence of this, the algorithm we propose by which clients estimate real time allows key distribution to proceed securely even during a lengthy unavailability of the time service. This has allowed us to explicitly *not* replicate the time service so that it will be easier to protect, and to achieve resilience to a time service unavailability through the client algorithm for estimating time.

2.1 The algorithm

Clients interact with the time service by the simple RPC-style protocol shown in figure 1. We assume that the time server possesses a private key $K_{\mathcal{T}}$ whose corresponding public key is well-known. (There is a similar shared-key protocol.) At regular intervals, a client queries the time service with a *nonce identifier* N [NS78], a new, unpredictable value. The time server immediately replies with $\{N, T\}_{K_{\mathcal{T}}}$, i.e., its current time value T and the nonce, both signed with $K_{\mathcal{T}}$. The client considers the reply valid if it contains N and can be verified with the public key of the time service.

Figure 1: Protocol by which C interacts with time service $\mathcal{T}$.

$$C \rightarrow \mathcal{T}: \quad N$$
$$\mathcal{T} \rightarrow C: \quad \{N, T\}_{K_{\mathcal{T}}}$$

The method by which a client uses this reply assumes that the client has access to a local hardware clock H that measures the length $t - t'$ of a real time interval $[t', t]$ with an error of at most $\rho(t - t')$ where $0 \leq \rho < 1$. That is,

$$(1 - \rho)(t - t') \leq H(t) - H(t') \leq (1 + \rho)(t - t'). \tag{1}$$

We assume that the time server's clock is perfectly synchronized to real time. (For all practical purposes, this can be done by attaching a WWV receiver or a very accurate clock to the time server's processor via a dedicated bus.) There are also assumed to be known, minimum real-time delays min_1 and min_2 experienced, respectively, between when a client initiates a request to the time service and when the time server receives that request, and between when the server reads its local clock value and the reply is verified as authentic at the client. (In our implementation, min_2 is substantially longer than min_1, because it includes the delays for signing and verifying the response.)

It is not difficult to prove that immediately after a client receives and verifies a response from the time service, it can characterize the current real time $\hat{t}$ by:

$$\hat{t} \in [T + min_2,\ T + r/(1-\rho) - min_1], \tag{2}$$

where T is the timestamp in the reply and r is the round trip time measured by the client, beginning when it sent the request and ending after it verified the reply. By combining (1) and (2), the client can characterize any later time $t \geq \hat{t}$ by:

$$t \in [L(t),\ U(t)], \tag{3}$$

where

$$L(t) = (H(t) - H(\hat{t}))/(1+\rho) + T + min_2$$

and

$$U(t) = (H(t) - H(\hat{t}))/(1-\rho) + T + r/(1-\rho) - min_1.$$

To estimate the time, the client uses either $L(t)$ or $U(t)$, depending on which is more conservative. In particular, it accepts an authentication protocol message as valid at time t only if the timestamp in the message plus the message lifetime is greater than $U(t)$. However, when timestamping a message to allow others to detect a later replay of that message, it obtains the timestamp from $L(t)$. The benefit of this scheme is that it is *fail-safe* [SS75]: a message with lifetime Δ sent by a (correct) client at time t will never be accepted by another client after time $t + \Delta$.

Clients periodically resynchronize with the time service, in order to narrow the interval (3). A successful resynchronization results in new values of $H(\hat{t})$, r and T for the calculation of $U(t)$ and $L(t)$. Resynchronization attempts can fail, however, when the round trip time r for the attempt exceeds some timeout value. When this happens, the client continues to attempt to resynchronize with the service at regular intervals, while maintaining the values of T, r and $H(\hat{t})$ obtained in the last successful resynchronization for the calculation of $L(t)$ and $U(t)$. Thus, if the time service becomes unavailable, the interval (3) at each client will drift wider with time. If the time service is unavailable for too long, eventually principals' values of $U(t)$ will exceed their values of $L(t)$ by the protocol message lifetimes, and all messages will be perceived as expired immediately upon creation.

While this bounds the amount of time that the system can continue to operate without the time service, calculations in our system indicate that this bound is not very tight: for most reasonable system parameter settings this bound is several days, and can be made much larger through further tuning. These calculations, and preliminary tests in our system, lead us to believe that the system, if tuned correctly, should be able to operate without the time service for sufficiently long to restart the time service, even if the restart requires operator intervention. More comprehensive testing of this hypothesis is currently underway.

2.2 Comparison to alternative designs

We derived our algorithm from that presented in [Cri89] for implementing a time service. The primary difference between ours and that in [Cri89] lies in how clients use the interval (2). In the latter, the client uses the midpoint of (2) as its estimate of the time at time t, as this choice minimizes the maximum possible error, and the client estimates future times as an offset, equal to the measured time since the last resynchronization, from this midpoint.[1] However, like any other clock synchronization algorithm in which each client maintains a single clock value, this algorithm is not fail-safe: e.g., if the midpoint of (2) were too low, then the client's future estimates of the time would tend to be low, and thus expired messages may be incorrectly accepted. We feel that our approach, which is fail-safe, is better for our purposes.

A reasonable alternative to not replicating our time service is to replicate it in such a way that it would provide a correct service despite some server failures and corruptions. For instance, a client could use the robust averaging algorithm of [Mar90] to obtain an interval of bounded inaccuracy containing real time from a set of n time servers, provided that fewer than $\lfloor n/3 \rfloor$ servers are faulty or corrupt. Nevertheless, such approaches place larger burdens on the administrator of the service than does ours, because the administrator must protect multiple servers, instead of only one, to ensure the integrity of the service. Since the availability of the time service is not crucial, this burden and the additional costs of replication are difficult to justify.

Numerous other approaches to clock synchronization have been proposed, but for brevity, we do not discuss them all here. Unlike ours, however, most assume upper bounds on message transmission times, and to our knowledge, none provide a fail-safe algorithm for estimating time in authentication protocols. We thus believe that our approach is unique in providing this property under relatively few assumptions.

3 The authentication service

Like those in [TA91, LABW91], our authentication service is of the public key variety, that produces public key *certificates* for principals. Each certificate $\{P, T, K_P^{-1}\}_{K_A}$ contains the identifier P of the principal, the public key K_P^{-1} of the principal, and the *expiration* time T of the certificate, all signed by the private key K_A of the authentication service. A principal uses these certificates to map principal identifiers to public keys, by which those principals (who presumably possess the corresponding private keys) can be authenticated; the details are discussed in [LABW91]. In general, a principal can request the certificate for any principal from the authentication service.

The need for security in such an authentication service is obvious: as the undisputed authority on what public key belongs to what principal, the authentication service, if corrupted, could create

[1] This is a simplification of the algorithm in [Cri89]; the actual algorithm also takes measures to ensure that client clocks are continuous and monotonic. These features, however, are unimportant for our purposes.

public key certificates arbitrarily and thus render secure communication impossible. It would also appear that, unlike the time service, the authentication service must be highly available, as its unavailability would prevent certificates from being refreshed when they expire. Other researchers have also noted that both security and availability, and thus the conflict between them, must be dealt with in the construction of authentication services (e.g., [LABW91, Gon93]). We first describe our method of balancing this tradeoff, and then compare it to other alternatives.

3.1 The algorithm

In [RB92], we describe a technique for securely replicating any service that can be modeled as a state machine. The technique is similar to regular state machine replication [Sch90], in which a client sends its request to *all* servers and accepts the response that it receives from a majority of them. In this way, if a majority of the servers is correct, then the response obtained by the client is correct. The approach in [RB92] provides similar guarantees but differs by freeing the client from authenticating the responses of all servers. Instead, the client is required to possess only one public key for the service and to authenticate only one response, just as if the service were not replicated.

We have constructed our authentication service using this technique. In its full generality, the system administrator can choose any *threshold value* k and create any number $n \geq k$ of authentication servers such that the service has the following properties:

1. Integrity: if fewer than k servers are corrupt, the contents of any properly signed certificate produced by the service were endorsed by some correct server, and

2. Availability: if at least k servers are correct, the service produces properly signed certificates.

As indicated above, a natural choice for the threshold value is $k = \lfloor n/2 + 1 \rfloor$, so that a majority of correct servers ensures both the availability and the integrity of the service.

Our technique employs a *threshold signature scheme*. Informally, a (k, n)-threshold signature scheme is a method of generating a public key and n *shares* of the corresponding private key in such a way that for any message m, each share can be used to produce a *partial result* from m, where any k of these partial results can be combined into the private key signature for m. Moreover, knowledge of k shares should be necessary to sign m, in the sense that without the private key it should be computationally infeasible to

1. create the signature for m without k partial results for m,

2. compute a partial result for m without the corresponding share, or

3. compute a share or the private key without k other shares.

The replication technique does not rely on any particular threshold signature scheme. For our authentication service, we have implemented the one in [DF92], which is based upon RSA [RSA78].

Given a (k, n)-threshold signature scheme, we implement our authentication service as follows. Let $\mathcal{A} = \{AS_1, \ldots, AS_n\}$ be the set of servers. Each authentication server AS_i, when started, is given the i-th share of the authentication service's private key $K_{\mathcal{A}}$, its own private key K_{AS_i}, the public key corresponding to the private key K_{AS_j} of each server AS_j, and the public keys for all principals. It is also given the public key of the time service to synchronize its clock as in section 2.

The protocol by which clients obtain certificates from the authentication service is illustrated in figure 2. A client C requests a certificate for a principal P by sending the identifier for P and a timestamp T to the servers. The purpose of T is to give the servers a common base time from which to compute the expiration time of the certificate;[2] we discuss how C chooses T below. When each server AS_i receives the request, it extracts T and tests if T is no more than its current value of $L(t)$. If so, it produces its partial result $pr_i(P, T+\Delta, K_P^{-1})$ for the contents $(P, T+\Delta, K_P^{-1})$ of P's certificate, where Δ is the predetermined lifetime of the certificate. AS_i then sends $pr_i(P, T+\Delta, K_P^{-1})$ to the other servers, signed under its own private key. When it has authenticated $k-1$ other partial results from which it can create the certificate $\{P, T+\Delta, K_P^{-1}\}_{K_{\mathcal{A}}}$, it sends the certificate to C.

Figure 2: Protocol by which C obtains a certificate for P.

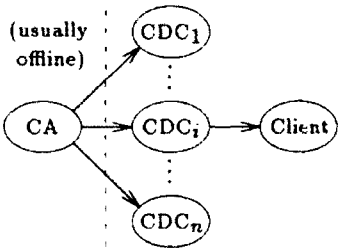
$$
\begin{aligned}
C \rightarrow \mathcal{A}: &\quad P, T \\
(\forall i)\ AS_i \rightarrow \mathcal{A}: &\quad \{P, T+\Delta, pr_i(P, T+\Delta, K_P^{-1})\}_{K_{AS_i}} \\
(\forall i)\ AS_i \rightarrow C: &\quad \{P, T+\Delta, K_P^{-1}\}_{K_{\mathcal{A}}}
\end{aligned}
$$

Because each AS_i produces a partial result only if T is no more than its value of $L(t)$, where t is the time at which it receives the request, any certificate produced for this request has an expiration timestamp of at most $t + \Delta$ (assuming fewer than k corrupt servers). A client accepts a certificate at some time t only if the certificate expiration time is greater than the client's value of $U(t)$, which ensures that the certificate expiration time has not been reached.

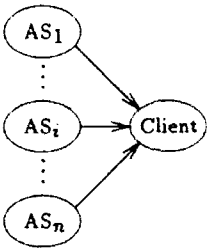
To obtain certificates of a maximum effective lifetime, a client should choose T to be close to (but less than) what it anticipates will be the correct servers' values of $L(t)$ when the request is received. In practice, it works well to have a client, when sending a request at time t, to set T to its own value of $L(t)$ minus a small offset $\delta \geq 0$, and to increase δ on subsequent requests if prior attempts to obtain a certificate failed. Because an unavailability of the time service will generally cause clients' values of $L(t)$ to drift from those of the servers, during a lengthy unavailability a client may need to set δ to several seconds to obtain a certificate, at the cost of reducing the effective lifetime of the certificate by that amount. However, since certificate lifetimes are typically at least several minutes, this would normally reduce the effective lifetime by only a small fraction.

[2] In a prior version of this protocol, each server used its value of $L(t)$ when the request was received as the base to compute the expiration time. This version was more sensitive to clock drifts and variances in request delivery times.

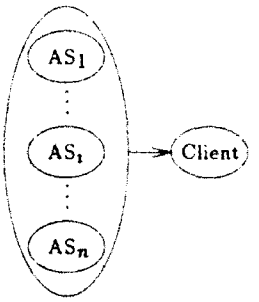
Figure 3: Design alternatives for a fault-tolerant public key authentication service.



(a) Offline certification authority (CA) [TA91, LABW91]. CA deposits long-lived certificates in an online, replicated certificate distribution center (CDC).



(b) State machine replication [Sch90]. Client authenticates certificate from each server and accepts key, if any, that occurs in a majority of certificates.



(c) Transparent state machine replication [RB92]. Client authenticates single certificate that could have been created only by a majority of servers.

3.2 Comparison to alternative designs

As previously mentioned, we are not the first to notice that the conflict between security and availability is evident in the construction of authentication services. In [Gon93], Gong proposed a method for dealing with this tradeoff in shared key authentication services such as Kerberos [SNS88]. Lampson, et.al., [LABW91] described a different solution that is appropriate for a public key authentication service similar to ours, which they call a *certification authority*.

In the latter solution, which is also implemented in SPX [TA91], the certification authority is usually offline, where it can be more easily protected by physical means (figure 3a). So its limited availability is not problematic, it produces long-lived certificates that are stored in an online *certificate distribution center* (CDC), which can be replicated for high availability [TA91]. Certificates are obtained only from CDC replicas, so if necessary, a certificate can be revoked by deleting it from all replicas. Thus, a client accepts a certificate only if both the highly secure certification authority and a CDC replica endorse it. The disadvantage of this scheme, noted by Lampson, et.al., is that the corruption of a CDC replica could delay the revocation of a certificate.

This problem could be addressed by using the technique of [RB92] to *securely* replicate the CDC. However, our approach of securely replicating the authentication service itself (figure 3c) addresses this problem more directly. Since the authentication service is online, it can refresh certificates frequently and create them with short lifetimes. Thus, the window of vulnerability between the disclosure of a principal's private key and the expiration of the principal's certificates can be greatly shortened, making revocation less crucial. Also, the service is both highly available and defensible, as it provides a correct service despite a minority of server failures and corruptions.

Our technique also has advantages over state machine replication [Sch90] (figure 3b) of the authentication service (or the CDC of [TA91, LABW91]). First, our approach requires less state at the client: the client needs only a single public key for the service, and need not maintain secure channels to the servers or retain any other replies but the first properly signed one. The last of these is especially beneficial if a principal obtains and forwards its own certificate to its partners in cryptographic protocols, as in the "push" technique described in [LABW91]. If the service were implemented using state machine replication, the principal would need to collect and forward a number of certificates equal to the size of a majority of the servers. Second, our technique requires the client to authenticate less communication, as the client can ignore all replies after the first properly signed one. Third, in our approach the configuration of the authentication service is largely transparent to clients, and so servers can be added or removed more easily.

There is, however, at least one disadvantage of our scheme with respect to the others mentioned here: due to the round of server communication, a client is likely to wait longer for a response in our scheme. As will be illustrated in section 4, though, in many situations communication with the authentication service can be performed in the background, off the critical path of any other protocol or computation, and in advance of any actual need for a certificate. (The certificate so obtained is then cached until the need for it arises.)

4 Utilization in a security architecture for fault-tolerant systems

As mentioned in section 1, our authentication and time services have been implemented as part of a more comprehensive security architecture for fault-tolerant systems [RBG92]. The architecture was developed in an effort to integrate security into Isis, a system for building fault-tolerant applications [BJ87, BSS91], although in principle it could be applied in other group-oriented systems such as V [CZ85] and Amoeba [KT91]. In this section, we briefly outline the architecture as it was designed and implemented for Isis,[3] and then focus on the roles played by the authentication and time services.

The basic abstraction provided by Isis is the *process group*, which is a collection of processes with an associated group address. Groups may overlap arbitrarily, and processes may create and join groups at any time. Processes communicate primarily by *group multicast*, i.e., by multicasting a message to the entire membership of a group of which it is a member. Isis further supports the model of *virtual synchrony*, so that message deliveries and notification of group membership changes (i.e., changes to the *group view*) appear in the same order at all group members.

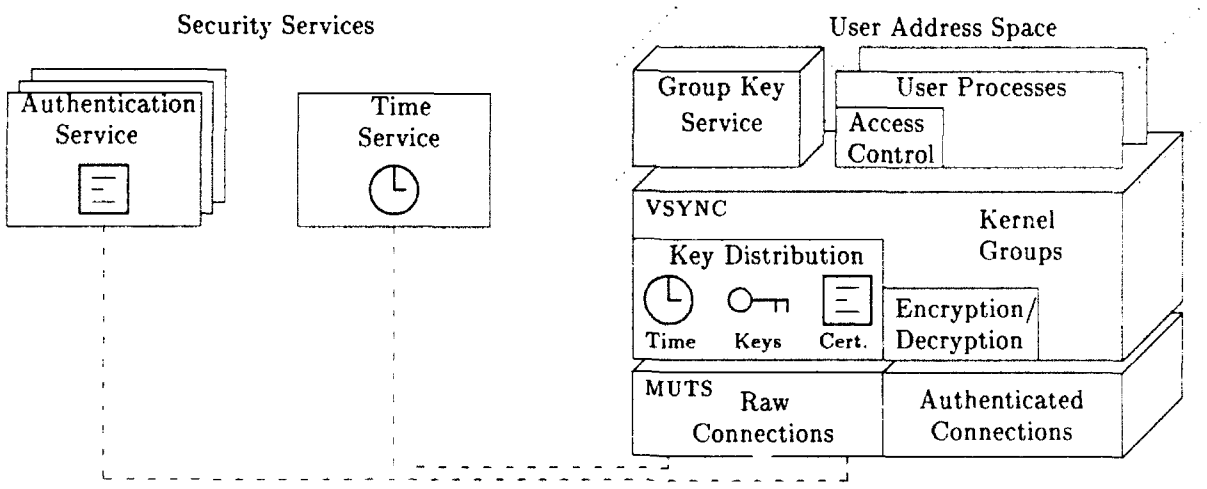
The security architecture makes the Isis programming model robust against malicious attack, while leaving the model itself unchanged. First, during group joins, the group and the joining process are mutually authenticated to one another. Second, a group member must explicitly grant

[3] More precisely, the security architecture was implemented as part of a new Isis system, also called Horus. Those familiar with earlier versions of Isis will notice that the new system architecture is very different from the previous.

each group join before the join is allowed to proceed. Third, the integrity of these mechanisms and the Isis abstractions, as well as the authenticity (and secrecy, if desired) of group communication, are guaranteed within each group that has admitted no processes on corrupted sites (i.e., sites at which an intruder has tampered with the hardware or operating system). These guarantees are achieved with minimal changes to the process group interface. So, existing Isis tools and applications can execute "securely" with little or no modification.

The security architecture as implemented in Isis is illustrated in figure 4. On each site, the core Isis functionality is implemented in a transport layer entity called MUTS and a session layer entity called VSYNC, both of which reside in the operating system kernel [vRBC+92]. The purpose of MUTS is to provide reliable, sequenced multicast among sites; VSYNC then implements the process group and virtual synchrony abstractions over this service. The security architecture augments these layers with protocols for distributing and using *group keys*. Each group has a pair of keys (the group keys) that are shared by all sites in the group.[4] One is a key to a symmetric cipher. This is used by MUTS to establish authenticated connections within the group, which preserve the authenticity and order of all internal group communication, and by VSYNC to distribute keys for the encryption of user messages. The other is an RSA private key whose corresponding public key is incorporated into the group address; this allows any process with the group address to authenticate group members. More detail about the use of group keys and group addresses can be found in [RBG92].

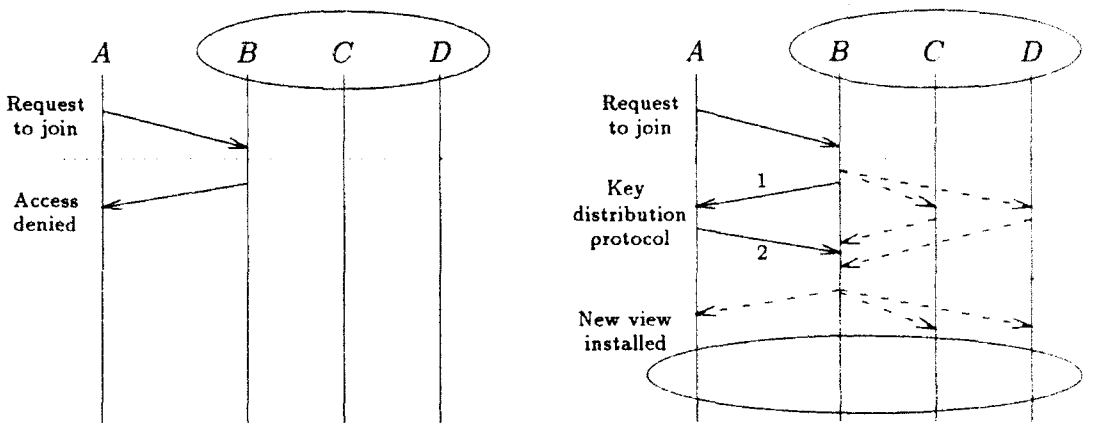
Figure 4: The Isis security architecture.



[4]The security dangers of replication also apply to group keys. However, we view this replication as acceptable at this level of the system for two reasons. First, the user has complete control over where the group keys are replicated. Thus, he or she can make this determination based upon the particular application and environment. Second, the disclosure of a group's keys corrupts only that group, not the entire system.

Group keys are created by a user-level operating system service called the *group key service*. It generates sets of group keys in the background and caches them in VSYNC. (This is done to remove the costly generation of an RSA key pair from the critical path of group creations.) When a local process requests to create a group, VSYNC removes a set of group keys from its local cache and associates them with the group. When VSYNC receives a join request for this group, it makes an upcall to a local member, which either grants or denies the request. If granted, VSYNC communicates the group keys to the joining site, using a Needham-Schroeder style protocol [NS78]. (See figure 5.)

Figure 5: Overview of the VSYNC group join protocol.



(a) A process on site A requests to join the group containing (processes on) sites B, C, and D. A sends the request to B, which delivers the request to its local member. Here, the member denies access, and B replies accordingly.

(b) In this case, access is granted. The group keys are securely sent to A in message 1, in parallel with a group synchronization protocol. After the keys are acknowledged (message 2), the new group view is installed.

It is this group join protocol that depends upon our authentication and time services. Each site is booted with the public keys of these services and its own private key; the authentication service possesses the corresponding public key. VSYNC periodically synchronizes with the time service and obtains (and periodically refreshes) a certificate for its site from the authentication service. When a local process asks to join a group, VSYNC forms a request including its current value of $L(t)$, signs the request with its site's private key, and prepends the certificate to the request before sending. A site that receives this message can authenticate the requesting site by checking the authentication service's signature on the certificate, extracting the public key from the certificate, and then checking the signature on the request. Moreover, it can verify the timeliness of the message by comparing the timestamps in the certificate and the request to its current value of $U(t)$, as described previously. For brevity we omit further description of the protocol; the techniques are well-known [BAN89].

There are two points worth emphasizing about our use of the authentication and time services. First, neither service is on the critical path of the group join protocol. This is more notable in the case of the authentication service, because in most systems, the authentication service (or, in [TA91, LABW91], the CDC) *is* on the critical path of authentication protocols. Second, the transparency of replication in the authentication service simplifies the protocol. If, e.g., state machine replication were used, each site would need to maintain certificates from a majority of servers to prepend to its requests, and these certificates would need to be refreshed on a per-server basis. This would also result in a substantial computational overhead for authenticating these certificates.

We leave discussion of other performance issues in the system to a companion paper [vRBR93]. Performance has, however, been a major goal of our activity. Several aspects of our architecture, such as cryptographic message protection and the key exchange component of the group join protocol, have performance implications. There are also synchronization costs for some styles of group communication. Nevertheless, we have reduced many of these costs using the same techniques used to reduce the costs of interacting with the authentication and time services, namely moving expensive operations off critical paths and caching information extensively. Overall, the performance impact of our work is negligible for a large class of Isis applications, and modest even in worst-case situations.

5 Summary and discussion

In this paper we have presented the design and implementation of an authentication service and a time service, with close attention to the tradeoffs between availability and security experienced in each. We have chosen to replicate only the authentication service, because unlike the time service, it *needs* to be replicated for availability. To compensate for this, it is built to tolerate a minority of server corruptions and failures. This places larger burdens on both the attacker and the defender than if the service were not replicated: the attacker must corrupt more servers to corrupt the service, but the defender must defend more servers to ensure the integrity and availability of the service.

The time service is not replicated, so that it is easier to protect. Moreover, its unavailability does not result in security breaches or hinder clients that continue to operate correctly. In fact, it could be temporarily taken offline for further protection if the need arises. While techniques exist for replicating time services so that some number of server corruptions could be tolerated, we have found that the additional costs of replication are difficult to justify.

Together these services form the core of a more comprehensive security architecture for fault-tolerant systems. In this role, they fault-tolerantly support the secure distribution of cryptographic keys that are used to protect communication within process groups. A beneficial feature of the services is that they appear nonreplicated to clients. This has resulted in simple protocols and clean algorithms for using the services, as well as mild storage and computation costs at clients.

We have fully implemented all of the mechanisms presented in this paper. There are, however,

issues that we have not yet fully addressed. For instance, these services in their current form cannot scale to very large systems, for both security and performance reasons. In a very large system, the services may become overwhelmed, and there may not be a single authority trusted to protect them. To alleviate this, services could be employed on a per-organization basis (as in a Kerberos realm [SNS88]) or hierarchically (as in [LABW91]). An alternative deployment of the authentication service would be to place each organization in charge of a different server.

Acknowledgements

We are grateful to Brad Glade for commenting on an early version of this paper, and especially for suggesting the inclusion of the timestamp in the first message of the protocol for obtaining certificates from the authentication service. We are also thankful to Fred Schneider and Tushar Chandra for providing comments on earlier versions of this paper.

References

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, Digital Equipment Corporation Systems Research Center, February 1989.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computing Systems*, 5(1):47–76, February 1987.

[BM90] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. *Computer Communications Review*, 20(5):119–132, October 1990.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computing Systems*, 9(3):272–314, August 1991.

[Cri89] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[CZ85] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computing Systems*, 3(2):77–107, May 1985.

[DF92] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91 Proceedings, Lecture Notes in Computer Science 576*, pages 457–469. Springer-Verlag, 1992.

[DS81] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[Gon92] L. Gong. A security risk of depending on synchronized clocks. *ACM Operating Systems Review*, 26(1):49–53, January 1992.

[Gon93] L. Gong. Increasing availability and security of an authentication service. To appear in *IEEE Journal on Selected Areas in Communications*, 1993.

[GZ84] R. Gusella and S. Zatti. TEMPO—A network time controller for a distributed Berkeley UNIX system. In *Proceedings of the USENIX Summer Conference*, pages 78–85, June 1984.

[HT88] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. In C. Pomerance, editor, *Advances in Cryptology—CRYPTO '87 Proceedings, Lecture Notes in Computer Science 293*, pages 379–391. Springer-Verlag, 1988.

[KT91] F. M. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[LABW91] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 165–182, October 1991.

[Mar90] K. Marzullo. Tolerating failures of continuous-valued sensors. *ACM Transactions on Computing Systems*, 8(4):284–304, November 1990.

[Mil89] D. L. Mills. Network Time Protocol (version 2) specification and implementation. RFC 1119, Network Working Group, September 1989.

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[OR87] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating Systems Review*, 21(1):8–11, January 1987.

[RB92] M. K. Reiter and K. P. Birman. How to securely replicate services. Technical Report 92-1287, Department of Computer Science, Cornell University, June 1992. Submitted to *ACM Transactions on Programming Languages and Systems.*

[RBG92] M. K. Reiter, K. P. Birman, and L. Gong. Integrating security in a group oriented distributed system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 18–32, May 1992.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SNS88] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, February 1988.

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[TA91] J. J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 232–244, May 1991.

[VK83] V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.

[vRBC+92] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX Microkernels and Other Kernel Architectures Workshop*, April 1992.

[vRBR93] R. van Renesse, K. Birman, and M. Reiter. Architecture and performance of the Horus system. In preparation, 1993.